IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4286

# Implementing the Event Sharing Paradigm: The Multipoint Event Sharing Service (MESS)

Erik Molenaar
Department of Computer Science 4, University of Technology Aachen, Germany, erik@molenaar.de

Dirk Trossen
Nokia Research Center Boston, Tel: (781) 993-3605, Dirk.Trossen@nokia.com



## ABSTRACT

*Shared collaboration between distributed users gains more importance due to the globalization of organizations and institutions. Beside exchanging audiovisual data, sharing spreadsheets or graphics is of utmost importance, especially in scenarios for tele-working or tele-education. Although the Internet has gained more ground in our daily work, most applications nowadays are not prepared for shared collaboration, and it is expected that this non-awareness of distribution will remain persistent for most of the applications. For that, application sharing technologies have been developed to encounter the problem sharing these kind of applications among a set of distributed users. Two different paradigms to realize application sharing can be distinguished, namely sharing the application's output or the application's evolving state. In this paper, the realization of an application sharing service is presented, based on the latter paradigm, which is mostly suited for closed development or teaching scenarios. The requirements for the service as well as its realization are outlined, together with the lessons we learned from this realization.*

## INTRODUCTION

For collaboration among a group of users, sharing audiovisual, textual, graphical, or even interface-related information is the essence of systems that realize *computer supported collaborative work* (CSCW). Several toolkits have been developed and studied in the past. Since most applications, being used in private and work life nowadays, are merely usable on the computer on which they are executed, collaboratively working with a single application is the most challenging part of CSCW. This is not only true because these applications are not aware that they are executed in a distributed environment, but in particular because of the numerous possibilities of data to be shared among the distributed users, when performing a local application.

Thus, the distribution of the application's functionality over the network has to be added transparently and, more important, subsequently without changing the application's semantic. The effect has to be created at each remote site that the application is running locally and therefore can also be controlled by any remote user with a more or less immediate effect to the application.

The realization of application sharing involves the synchronized transfer of application-specific data among users, and it faces several challenges to be solved [6]. The *number of interception points* is part of the indicator for overhead that is added to the system, together with the *amount of transferred data* per interception. Each application sharing technique intercepts the local system to gather the required information and to build an appropriate data packet to be distributed among the users. This packet has to be transferred through the local protocol stack. These actions degrade the overall system performance. As a consequence, a small number of interceptions is desired, while keeping the amount of transferred data per interception low.

Independence from the member's operating system, i.e., *heterogeneity* of the users' system, is crucial for a wide applicability of the technique. Moreover, *latecomer's support* should be provided without leading to inconsistencies of the application's state. To each shared application, input data is usually fed into to evolve in subsequent states. However, this *shared data problem* should not lead to inconsistencies of the distributed application instances. And eventually, the copies of the application have to be kept in *synchronization* to ensure consistency of the workspace among all users due to the different processing speed of the sites and the different transmission delays.

Two different paradigms can be distinguished to tackle the abovementioned challenges, namely *Output Sharing* and *Event Sharing*. In [6], a qualitative comparison of both paradigms is presented, outlining the different application scenarios for both paradigms. It was concluded that the latter is best suited for closed group environments with a limited set of input data to be shared. As a consequence, it seems to be a promising candidate for shared engineering [7], multimedia presentation, or tele-teaching scenarios. The event sharing paradigm is based on the assumption that if a set of identical applications is executed with the same start state and evolves using the same sequence of events, its timeline evolution is identical on each site. In the light of this assumption, the approach can be outlined as follows:

- grab the start state to be distributed among all group members
- start local copies of the application to be shared on each host
- distribute input events of the currently controlling user to evolve the current application's state

Although the approach seems fairly simple, coping with the abovementioned challenges is not at all an easy task. Especially the shared data problem and the determination of the start state of the application can be seen as the main challenges when realizing the paradigm.

This paper presents a realization of the event sharing paradigm, called *Multipoint Event Sharing Service* (MESS), outlining the architecture and the implementation issues to be addressed. For that, a component-based architecture is presented, which is mapped onto an object-oriented design to bring the system to life. The currently provided functionality and obtained performance is described, which is very encouraging, especially for the targeted application scenarios. However, the realization encounters several difficulties, which will be presented as the lessons we learned from our work.

The remainder of the paper is organized as follows. Section 2 defines the requirements for the presented work, while Section 3 outlines the architecture and realization of the application sharing service. Section 4 discusses lessons we learned, while Section 5 gives pointers for further reading. Section 6 eventually concludes the paper with a discussion of our future work.

## REQUIREMENTS

The main requirement for an application sharing service is to enable a synchronous view of an application on several participating computers. In [6], the idea was formulated to investigate the possibilities of applying the event sharing paradigm and the gain it can offer. In this work, a realization of an event sharing service is proposed. Its design will be a consequence of the requirements, presented in this section. Before outlining the requirements, some definitions and theoretical background are needed.

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference. 

### Definitions and Background

The *state* of an application describes the current snapshot of the application itself and all resources it addresses. *Resources* can be anything that is not the application itself, but is changed or used by the application to determine its behavior. Examples are files, registry entries, or the system time. Phenomena that change the application's state are called *events*.

A *stable state* of the shared application is given, if the execution behavior of all instances is equivalent. For example, if a menu entry is selected, the same action belonging to the corresponding menu entry should be performed on all machines. This stable state will sometimes be referred to as being in a *consistent* or *equivalent state.*

*Deterministic behavior* of an application means that if a set of this application is started in an equivalent state, and the same set of events is presented to those instances, then the same state transitions will happen for all instances. It is important to realize that this definition of deterministic behavior is more relaxed as other definitions, in the sense that resources, that an application might need, are considered as part of the environment. Where other definitions might assume that an application is no longer behaving deterministically if e.g., the system time of the local machine is used, this definition regards the system time as a part of the environment.

Given an application that behaves deterministically, the following statement is valid:

*Theorem 1: A set of instances of an application that behaves deterministically can be held in a stable state if the starting state and all events can be captured.*

The proof of this statement is a simple induction: Assume all instances in stable state at the beginning. Since every event can be captured, these events are fed into each of the instances to initiate a transformation of state. Because of the definitions of state and event, these successor states are stable again from a viewpoint of a neutral observer somewhere in the session.

### Requirements

Apart from the major requirement that the application to be shared must behave deterministically, the following requirements for the application sharing service can be defined to keep the shared instances in stable state over the timeline:

- All participating instances must start in an equivalent state.
- During runtime of the session, all events that change the application's state must be captured and broadcast to all participants.
- If some events access resources, these must be provided to all participants.
- Synchronization of instances must be offered.

In addition to these functional requirements, the following minor requirements have to be addressed by a realization:

- An interface with the participant has to be offered.
- Distributed messages that are sent have to be marshaled, i.e., being transferred in a common syntax.
- Latecomer's support has to be addressed.

Since a shared application service is using resources from existing conferencing systems, such as [5], the following requirements for this part of the system can be derived:

- Conference management, i.e., joining and leaving conferences, should be provided.
- Floor control is needed to prevent multiple participants to control the application simultaneously.
- Reliable message transport shall be provided with global ordering of messages.
- If possible, multicast capabilities shall be utilized.

The abovementioned requirements will be used as a foundation for the design of an application sharing service based on the event sharing paradigm.

## MULTIPOINT EVENT SHARING SERVICE (MESS)

Based on the abovementioned requirements, the *Multipoint Event Sharing Service* (MESS) is presented to realize the event sharing paradigm.

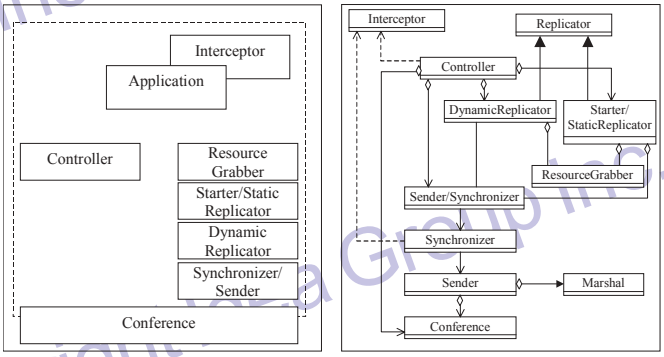
### Architecture

Figure 1 left shows the components of the MESS architecture, reflecting the practical proof of Theorem 1, i.e., the concept of starting in an equivalent state and evolving during runtime of the session.

This concept is reflected by the *Starter/Static Replicator* and the *Dynamic Replicator* components. For these components to function, they need some utility components. At the currently controlling application side, the *Interceptor* gathers required event information. The resources that are used by these events are recognized through the *Resource Grabber*. The actual sending of both the needed resources and the events is prepared by the *Sender/Synchronizer*. This component also takes care of the synchronization and offers latecomer support. The required conferencing and data transmission functionality is provided by the *Conferencing* component, and the interaction with the participant and coordination of components are performed through the *Controller*.

This component architecture can be transformed in a UML framework, as shown in the right part of Figure 1. It is a straightforward mapping of the components onto classes with dedicated methods. This framework acts as the foundation of the actual implementation of the MESS architecture.

*Figure 1: MESS architecture*




The tasks of each component are described in more detail in the following sections.

*Controller*

The Controller has to start the service, implement the chosen policy for conference management, and provide a mechanism for a token management policy.

*Interceptor*

The Interceptor gathers required event information to be shared among the users and to be used for synchronization. Two kinds of events need to be handled. The first includes events originating from the user (*user events*). Examples are mouse movement, mouse buttons pressed, and keyboard keys pressed. The second type of events originates from the system. These *system events* have to be handled separately. As an example consider an application that renders and shows an animation. The animation speed will depend on the processing speed of each individual computer. If the event sharing service merely shared the user events, the participating computers would get more out of sync during runtime, since their speed is not the same, although

they were started synchronously. To cope with this effect, the progress in execution can be monitored and steered by system events. These events are not actually required to be shared since they are caused by the program execution as such, and therefore they should appear on all participating instances of the application. However, monitoring these events is required for synchronization.

*Resource Grabber*

The task of the Resource Grabber is to locate and identify all resources, including the application, on the controlling end-system and distribute this information among the group to ensure a consistent state of the application.

*Starter / Static Replicator*

This component takes care of all instances of the application to be in an equivalent state upon startup. It decides what resources will be distributed, and it takes care that the local settings for each participant are brought in a consistent state. For that, the input of the Resource Grabber is used.

*Dynamic Replicator*

The Dynamic Replicator is responsible for keeping all participating applications in an equivalent state after the session has started by appropriately sharing event information, determined by the Interceptor, among the group.

*Synchronizer / Sender*

The Synchronizer/Sender is responsible for synchronization and latecomer support. For synchronization of user events, these events are broadcast, while all application instances are halted locally. After successful delivery, the next user event is processed. For system events, the application is halted after *n* events have been counted. These events are not broadcast since they are generated by the system on each of the participating instances. Only after *n* system events have been processed on all instances, application progress is resumed. For latecomer support, a form of state dependent startup of the participating application is needed, which could be provided using a log file of previously occurred events.

*Conference Control*

This component deals as an interface to the underlying conferencing system, using functionality for conference management, floor control, and transport functionality.

### Realization

The proposed MESS architecture was implemented as a prototype to demonstrate the feasibility of the event sharing approach. Although the current design allows for sharing all types of events and resources, the actual implementation has to make certain tradeoffs to keep the realization simple and feasible, but also to demonstrate the potential of the proposal. As a naive approach, one could try to watch and share every thinkable resource. This is neither necessary nor desired. Instead, one has to make a tradeoff between maximizing the limitation in bandwidth and system overhead on one hand and to minimize the amount of applications that need services that are not implemented as a result of the first on the other hand.

As a consequence, no system events are shared at this time, and there is no synchronization among the participants. Moreover, resource distribution among the participants is not provided. The current demo application is merely meant to experiment with the distribution of user events and to test the resulting functionality of applications that are shared in such a primitive environment. This functionality reflects the most important part of the service, namely the evolution of the application's state, and therefore demonstrates the ability of the concept to provide application sharing for certain scenarios. However, the missing functionality is easily integrated, since it mostly deals with capturing additional events, and synchronizing these appropriately at the controlling site.

As demonstration scenarios, simple text-editing as well as rotation of complex 3D objects are performed. The latter in particular happens in shared engineering scenarios, as described in [7], and is well suited to demonstrate the potential since it generates certain graphical output. However, due to the missing synchronization functionality, the computers usually run out of sync after a certain timeframe, which demonstrates the necessity of this functionality, i.e., to slow down the faster end-system(s) appropriately.

## EVALUATION AND LESSONS LEARNED

The MESS architecture can be evaluated as regards to complexity, functionality as well as resulting performance.

The proposed components add certain *complexity* to each end-system. The variety of state information to be grabbed and distributed usually varies in modern operating systems, e.g., script files or registry settings. However, collecting this information can be realized at central points by intercepting appropriate system calls, e.g., for reading registry settings. Similar to state information, event interception can also be realized centrally by intercepting appropriate system calls. Thus, the added overhead to the operating system is usually fairly minimal and centralized.

Although the proposed MESS architecture provides application sharing *functionality* for any kind of application with deterministic behavior, the actual functionality highly depends on the maturity of chosen implementation detail. For that, a tradeoff has to be made between the set of supported applications and the chosen complexity. For instance, the demonstration application shows that synchronization is necessary for many scenarios, though surprisingly many scenarios can be covered with limited or even no synchronization at all.

*Performance* of the proposed architecture can be evaluated in two dimensions. First, the added overhead to the system due to the interception to gather and distribute event information is a major performance measure. For that, the demonstration application shows that this additional overhead is fairly small. However, adding more system events and resources to the pool of information certainly decreases the overall performance, although the transmitted information remains small. As a second measure, the bandwidth consumption of the service is of importance. An estimation for the bandwidth consumption during runtime can be made based on the text-editing demonstration. Assuming a reasonable amount of entered text, e.g., 250 characters per minute. Further, assume one sync event after each pressed key as a conservative approach. Thus, the bandwidth consumption would be less than 700 bits per second with an event size of 16 bytes and a synchronization message size of 2 bytes. In the example of rotating 3D objects, the overhead to the system and the consumed bandwidth is even smaller since user events are usually generated with a smaller frequency. However, the bandwidth consumed for distributing resource information heavily depends on the amount of gathered information and the application as such. The more resources are used, the more information has to be distributed, either during startup or runtime of the session.

As a summary, the most important lesson we learned was that the basic concept of event sharing works with an impressive speed by leveraging local processing speed for the application functionality. However, we also learned that the integration of some system events with an additional synchronization to cope with different processing speeds highly increases the spectrum of applications that can be used with the system.

## FURTHER READING

Most available application sharing systems implement the GUI sharing paradigm, of which many are based on the *X-Windows* system, comprised of a central server on which the application is executed. The application's output is redirected to *X Windows clients* for rendering. Extending this system to a multipoint scenario, as done in [1][4][8], enables a shared application system for cooperative working. How-

ever, floor control facilities have to be added for coordinated control, which was done in [1][4].

Despite the wide deployment of X Windows systems, their applicability is mainly restricted to Unix systems. Although X Windows client software is available for other platforms, the problem remains to share for instance MS Windows software on other platforms. Hence, the heterogeneity problem is only partially solved when using an X Windows system. To tackle this problem, the ITU proposed a protocol for multipoint application sharing [3], defining platform-independent rendering and interception functionality. The disadvantages of this approach are mainly its underlying shared GUI approach, and therefore the overhead on the server system, and the usage of an ineffective transport system, which is defined in the ITU T.120 standard.

The work in [2] realizes the event sharing paradigm by replicating the entire data workspace before starting the application copies. Dynamically including shared data is not supported. Synchronization among the different copies is ensured for every incoming event, leading to a significant overhead instead of using specific synchronization events for overhead reduction. Moreover, the event mapping and distribution is realized on a central server. Hence, the proposal follows a distributed application, but a centralized control approach.

## FUTURE WORK

The proposed MESS architecture allows for sharing start state and event evolution of applications among a set of local copies in a shared workspace scenario, i.e., it implements the event sharing paradigm. However, the functionality of our demonstrator is currently restricted for the sake of simplicity.

In our future work, this functionality is to be increased, starting with the synchronization functionality to cope with out-of-sync effects. Moreover, finding some optima for the applications that can be served by the MESS while keeping the used bandwidth to a minimum is a field of future work.

In addition to enriching functionality, more systematic evaluation scenarios have to be defined to become a clear view of the overhead added to the system. Moreover, the demonstration system is used within a project, realizing a workspace for shared engineering (see [7]).

## REFERENCES

[1] M. Altenhofen, et al.: *The BERKOM Multimedia Collaboration Service*, Proceedings ACM Multimedia, 1993

[2] M. C. Hao, J. S. Sventek: *Collaborative Design Using Your Favorite 3D Application*, Proceedings IEEE Conference on Concurrent Engineering, 1996

[3] ITU-T: *Multipoint Application Sharing*, ITU-T Recommendation T.128, 1998

[4] W. Minenko, J. Schweitzer: *An Advanced Application Sharing System for Synchronous Collaboration in Heterogeneous Environment*, SIGOIS Bulletin, vol.15 no.2, pp. 40-44, 1994

[5] D. Trossen: *Scalable Conferencing Support for Tightly-Coupled Environments: Services, Mechanisms, and Implementation Design*, Proceedings IEEE International Conference on Communications, 2000

[6] D. Trossen: *Application Sharing Technology: Sharing the Application or its GUI ?*, Proceedings IRMA, 2001

[7] D. Trossen, A. Schueppen, M. Wallbaum: *Shared Workspace for Collaborative Engineering*, to appear in Annals of Cases on Information Technology, Volume IV

[8] K. H. Wolf, K. Froitzheim, P. Schulthess: *Multimedia Application Sharing in a Heterogeneous Environment*, Proceedings ACM Multimedia, 1995